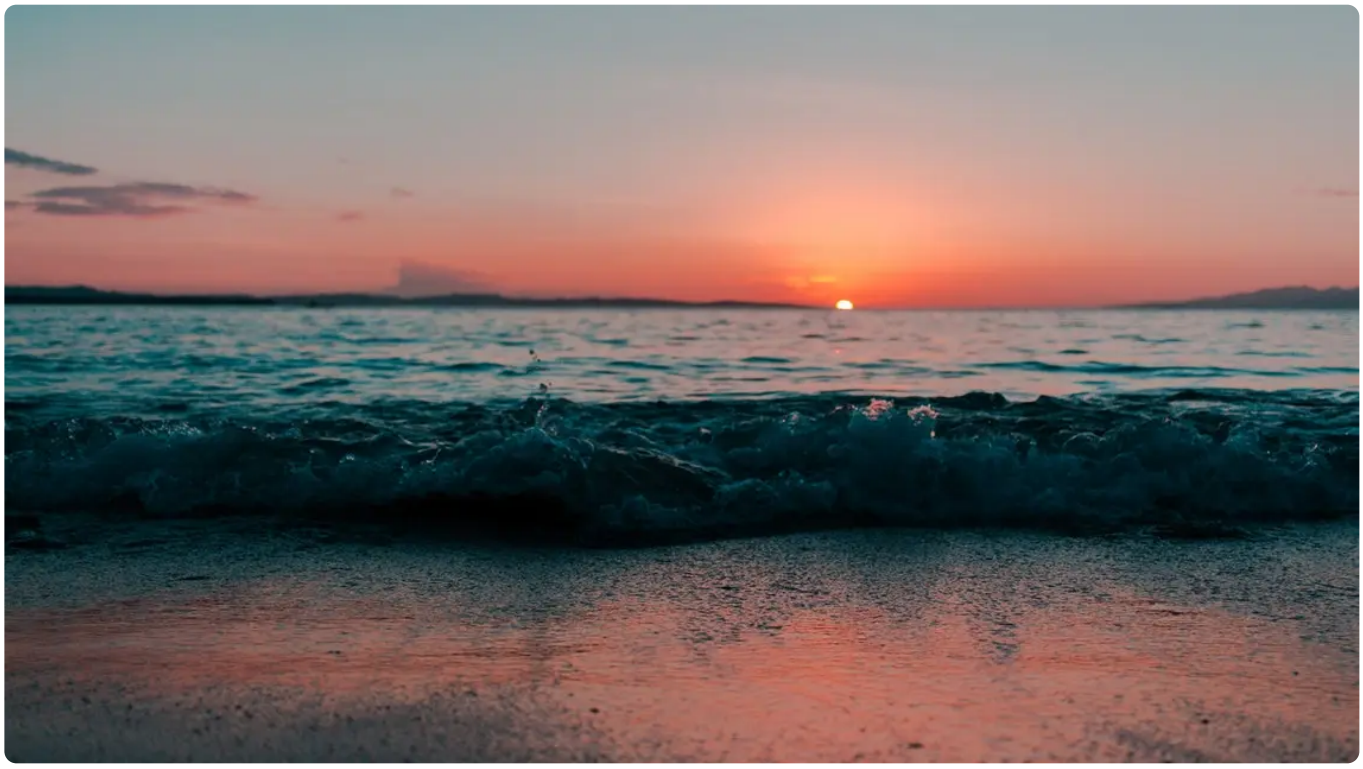
11 | 容器安全：如何使用 eBPF 增强容器安全？

2022-02-09 倪朋飞

《eBPF 核心技术与实战》 课程介绍 >



讲述：山荣
时长 12:58 大小 11.88M

你好，我是倪朋飞。

上一讲，我以最常见的网络丢包为例，带你一起梳理了 eBPF 所提供的网络功能特性，并教你使用 bpftrace 开发了一个跟踪内核网络协议栈的 eBPF 程序。虽然 eBPF 起源于网络过滤，并且网络过滤也是 eBPF 应用最为广泛的一个领域，但其实 eBPF 的应用已经远远超出了这一范畴。故障诊断、网络优化、安全控制、性能监控等，都已是 eBPF 的主战场。

随着容器和云原生技术的普及，由于容器天生共享内核的特性，容器的安全和隔离就是所有容器平台头上的“紧箍咒”。因此，如何快速定位容器安全问题，如何确保容器的隔离，以及如何预防容器安全漏洞等，是每个容器平台都需要解决的头号问题。

既然容器是共享内核的，这些安全问题的解决自然就可以从内核的角度进行考虑。除了容器自身所强依赖的命名空间、cgroups、Linux 权限控制 Capabilities 之外，可以动态跟踪和扩展内核的 eBPF 就成为了安全监控和安全控制的主要手段之一。 Sysdig、Aqua Security、Datadog 等业内知名的容器安全解决方案，都基于 eBPF 构建了丰富的安全特性。

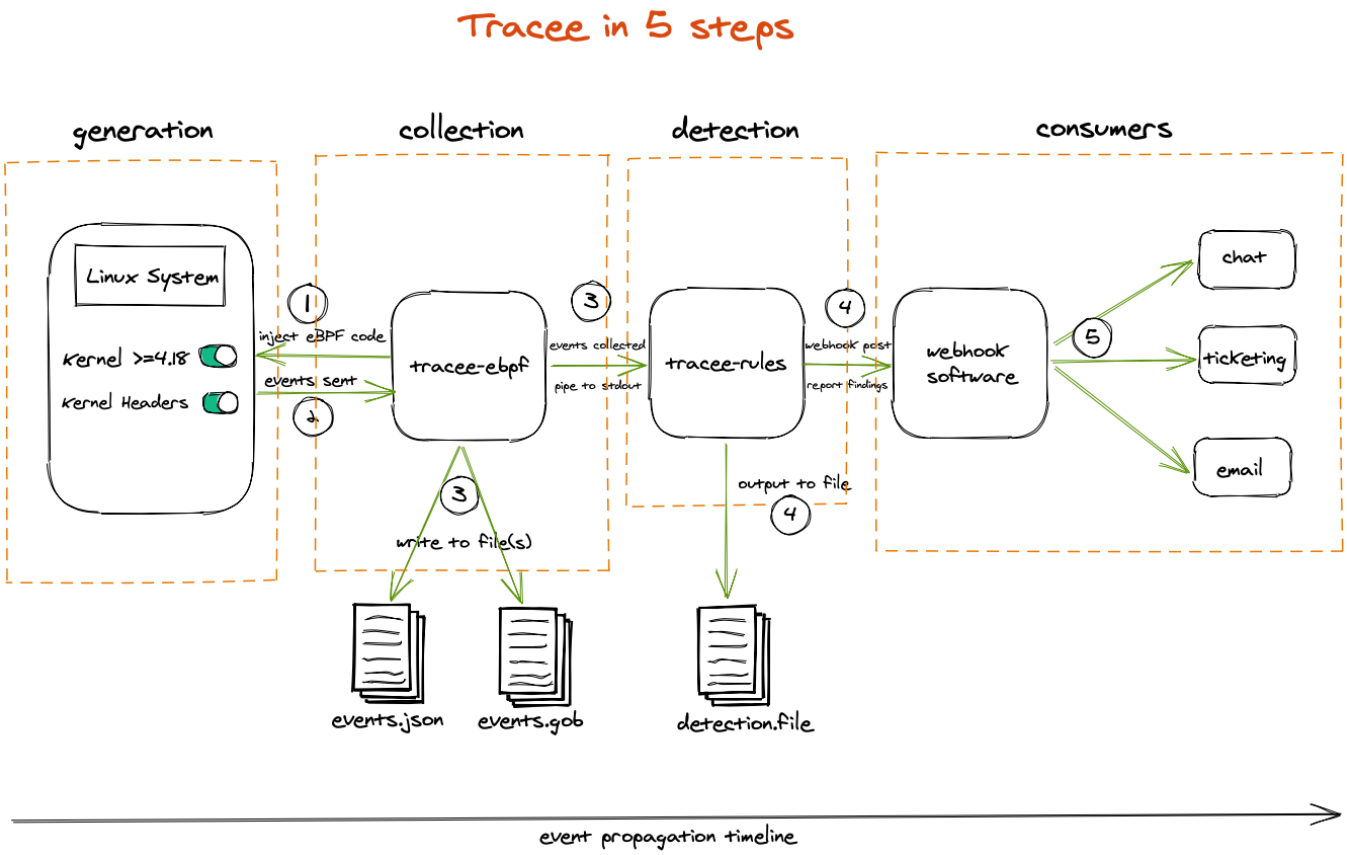
那么，到底如何使用 eBPF 来监控容器的安全问题，又如何使用 eBPF 阻止容器中的恶意行为呢？今天，我就带你一起来看看如何借助 eBPF 来增强容器安全。

eBPF 都有哪些安全能力？

安全这个词通常包含了非常广泛的任务，包括安全问题的分析与诊断、安全事件的检测，以及安全策略的执行等。针对这些任务，eBPF 又提供了哪些安全能力呢？

首先，对于**安全问题的分析与诊断**，eBPF 无需修改并重新编译内核和应用就可以动态分析内核及应用的行为。这在很多需要保留安全问题现场的情况下非常有用。特别是在紧急安全事件的处理过程中，eBPF 可以实时探测进程或内核中的可疑行为，进而帮你更快地定位安全问题的根源。

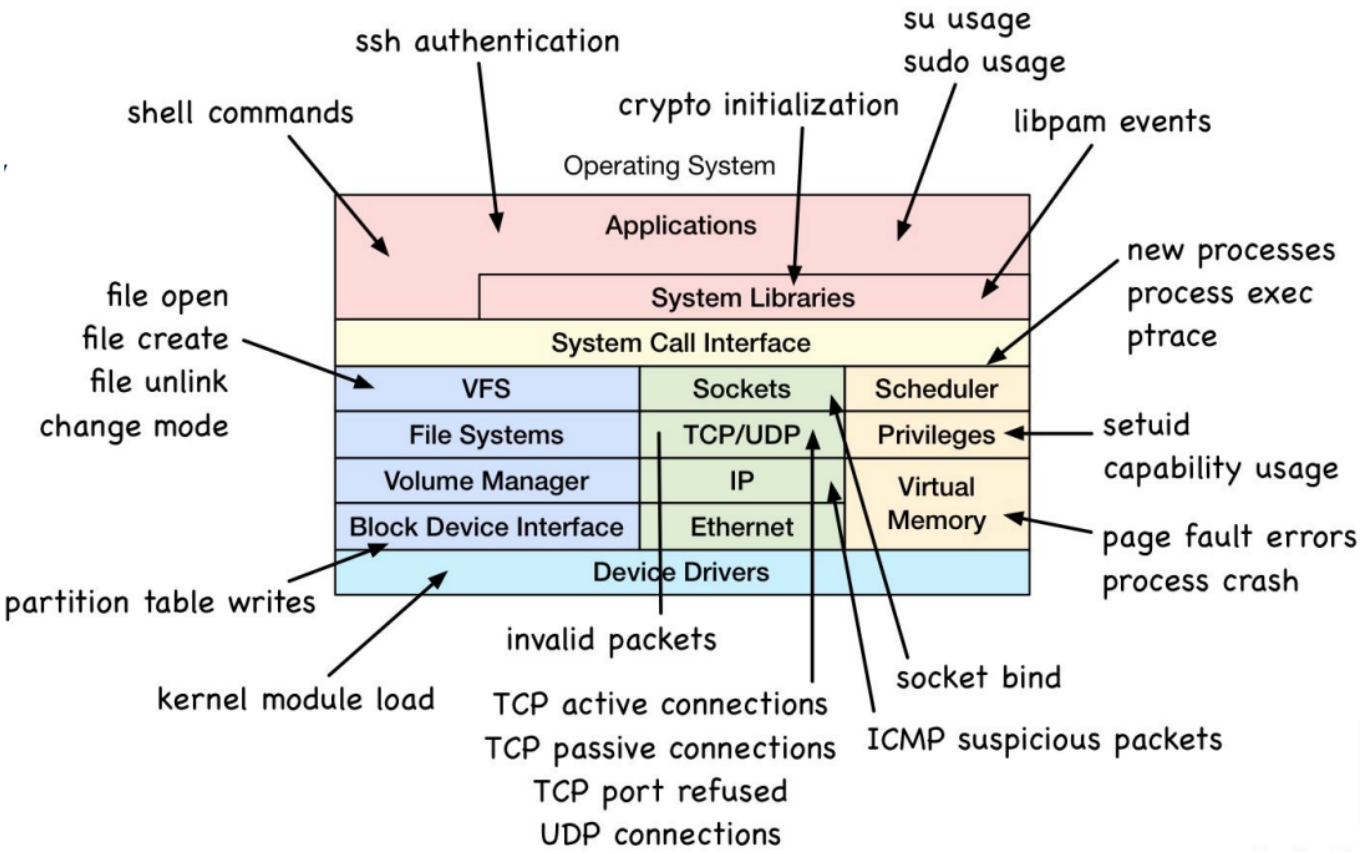
比如，Aqua Security 开源的 Tracee 项目就利用 eBPF，动态跟踪系统和应用的可疑行为模式，再与不断丰富的特征检测库进行匹配，就可以分析出容器应用中的安全问题。下面是 Tracee 的工作原理示意图（图片来自 Tracee 文档）：



Tracee 原理示意图

其次，对于**安全事件的监测**，eBPF 的事件触发机制不仅可以用极低的开销，动态监测内核和应用程序中的各类安全事件，更可以避免其他基于采样或统计机制的安全检测系统中经常发生的安全事件漏检问题。

如下图（图片来自 brendangregg.com ）所示，kprobe、uprobe、tracepoint、USDT 等各类探针已经非常全面地涵盖了从应用到内核中的各类安全问题相关的跟踪点：

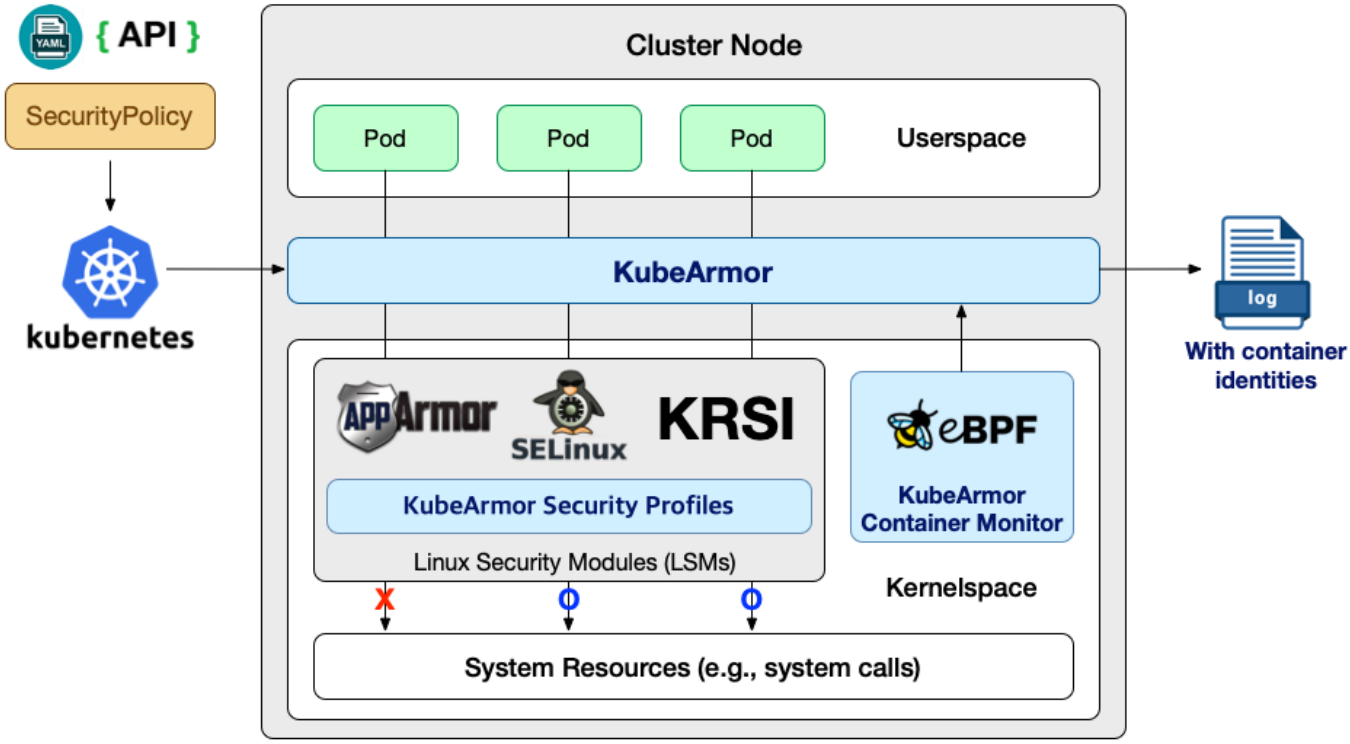


eBPF 安全跟踪点

比如，Sysdig 贡献给 CNCF 基金会的 Falco 项目，就利用 eBPF 在运行时监测内核和应用中是否发生了诸如特权提升、SHELL 命令执行、系统文件（比如 `/etc/passwd`）修改、SSH 登录等异常行为，再通过告警系统实时将这些安全事件及时推送给你。

最后，对于**安全策略的执行**，安全计算（seccomp）、Linux 安全模块（LSM）等 Linux 已有的安全机制，均可以通过 eBPF 来进行安全审计和决策执行，阻止系统和进程中的非法操作，甚至可以通过辅助函数 `bpf_send_signal()` 直接把有安全隐患的进程杀死。在网络安全方面，eBPF 不仅可以用来实时探测和跟踪网络请求，更可以在探测到网络攻击等危害行为时直接在网络协议栈之前丢弃数据包，从而实现极高的网络性能。

如下图所示（图片来自 KubeArmor 文档），容器运行时安全系统 KubeArmor 就利用 LSM 和 eBPF，限制容器中进程执行、文件访问、网络连接等各种违反安全策略的操作。



KubeArmor 工作原理

如何使用 eBPF 分析容器的安全问题？

既然容器还是共享内核的，运行在内核中的 eBPF 程序自然也能够跟踪和分析容器中的应用程序。但由于容器利用 Linux 的 namespace 机制进行了隔离，其跟踪和分析方法又跟直接运行在主机内的进程有些不同。

以跟踪恶意程序的执行为例，为了躲避安全监控，很多恶意程序并不是在容器一开始启动的时候就运行了恶意进程，而是先启动一个正常程序，之后再创建新的恶意进程。这种场景特别容易出现在容器安全漏洞被恶意软件侵入的场景。

那么，如何用 eBPF 来分析这类安全问题呢？我想，你可能已经想起来了，我们课程的 07 讲 其实已经实现了对新进程创建的跟踪程序，即跟踪系统调用 execve。比如，执行下面的 bpftrace 命令，就可以跟踪新创建的进程：

复制代码

```
1 sudo bpftrace -e 'tracepoint:syscalls:sys_enter_execve { printf("%-6d %-8s", p
```

打开一个新终端，执行一条 ls 命令，然后你就会看到如下的输出：

复制代码

```
1 8964   bash    ls --color=auto
```

接下来，参考 Docker 官方文档安装 Docker 之后，再执行下面的命令，启动一个 Ubuntu 容器：

复制代码

```
1 # -it表示进入容器终端，--rm表示终端关闭后自动清理容器
2 docker run -it --rm --name bash --hostname bash ubuntu:impish
```

在容器中执行 ls 命令，忽略容器启动过程中的进程跟踪信息（Docker 在启动容器过程中也会执行大量的命令），你会看到跟刚才类似的输出：

复制代码

```
1 9018   bash    ls --color=auto
```

这个输出跟刚才在主机中执行 ls 后的结果是一样的，只根据这个输出，我们显然没法区分 ls 是不是运行在容器中。

实际上，虽然所有容器都是共享内核的，但不同的容器之间还是通过命名空间进行了隔离。你可以使用 lsns 命令来查询容器或者主机的命名空间。比如，在刚才的容器终端中执行 lsns 命令，就可以看到如下的输出：

复制代码

```
1         NS TYPE   NPROCS PID USER COMMAND
2 4026531834 time         2   1 root bash
3 4026531835 cgroup       2   1 root bash
4 4026531837 user         2   1 root bash
5 4026532530 mnt          2   1 root bash
6 4026532531 uts          2   1 root bash
7 4026532532 ipc          2   1 root bash
8 4026532533 pid          2   1 root bash
9 4026532535 net          2   1 root bash
```

关于这些命名空间的含义，如果你还不了解，可以参考这里的 Linux 手册。

在内核中，进程的基本信息都保存在 task_struct 结构体中，其中也包括了包含命名空间信息的 nsproxy 结构体。nsproxy 结构体的定义如下所示：

复制代码

```
struct nsproxy {
  atomic_t count;
  struct uts_namespace *uts_ns;
  struct ipc_namespace *ipc_ns;
  struct mnt_namespace *mnt_ns;
  struct pid_namespace *pid_ns_for_children;
  struct net           *net_ns;
  struct time_namespace *time_ns;
  struct time_namespace *time_ns_for_children;
  struct cgroup_namespace *cgroup_ns;
};
```

为了区分一个进程是属于容器还是主机，我们可以在跟踪结果中输出 PID 命名空间和 UTS 命名空间中的主机名。

bpftrace 内置了表示进程结构体的 curtask，因而对前面的 bpftrace 脚本，我们可以进行下面的改进：

复制代码

```
tracepoint:syscalls:sys_enter_execve {
  /* 1. 获取task_struct结构体 */
  $task = (struct task_struct *)curtask;
  /* 2. 获取PID命名空间 */
  $pidns = $task->nsproxy->pid_ns_for_children->ns.inum;
  /* 3. 获取主机名 */
  $cname = $task->nsproxy->uts_ns->name.nodename;
  /* 4. 输出PID命名空间、主机名和进程基本信息 */
  printf("%-12ld %-8s %-6d %-6d %-8s", (uint64)$pidns, $cname, curtask->parent
}
```

这段代码中的具体内容含义如下：

第 1 处，把内置变量 curtask 转换为我们想要的 task_struct 结构体；

第 2 处，从进程信息的 nsproxy 中读取 PID 命名空间编号；

第 3 处，从进程信息的 nsproxy 中读取 UTS 命名空间的主机名（也就是在容器中执行 hostname 命令后的输出）；

第 4 处你已经非常熟悉了，就是把刚才获取的信息输出，以便我们观察。

由于这儿用到了很多内核数据结构，在运行之前，还需要给它引入相关数据结构定义的头文件：

复制代码

```
#include <linux/sched.h>
#include <linux/nsproxy.h>
#include <linux/utsname.h>
#include <linux/pid_namespace.h>
```

同时，由于输出的内容比较多，为了便于理解，你还可以在脚本运行开始的时候输出一个表头，表示每个输出的含义：

复制代码

```
BEGIN {
  printf("%-12s %-8s %-6s %-6s %-8s %s\n", "PIDNS", "CONTAINER", "PPID", "PID"
}
```

把头文件引入和改进后的 bpftrace 脚本保存到 execsnoop-container.bt 文件中（你也可以在 GitHub 上找到完整代码），然后打开一个新终端，运行下面的命令来执行：

复制代码

```
sudo bpftrace execsnoop-container.bt
```

接下来，分别在容器终端和主机终端中执行一个 ls 命令，就可以得到如下的输出：

复制代码

```
PIDNS        CONTAINER PPID    PID     COMM      ARGS
# 容器ls命令跟踪结果
4026532533   bash      41046   41335   bash      ls --color=auto

# 主机ls命令跟踪结果
4026531836   ubuntu.localdomain 40958   41356   bash      ls --color=auto
```

在输出中，容器 ls 命令跟踪结果中的 PID 命名空间 4026532533 跟上述容器中 lsns 结果是一致的，而主机名 bash 也跟运行容器时设置的 --hostname name 一致，因而我们很容易区分这条 ls 命令的来源。

你可以发现，**只要理解了容器的基本原理，在跟踪过程中加入容器的基本信息，容器内外进程的跟踪和分析并没有本质的区别。**

也许看到这里你会有疑问：这儿讲到的是内核态的跟踪，容器内外没有区别很正常。但如果是用户态的进程跟踪呢？你可以先思考一下，再继续下面的内容。

实际上，用户态进程的跟踪也是一样的，唯一需要注意的就是找到容器内二进制文件的正确路径。虽然容器文件系统在不同的 mount 命令空间中，但对于每个进程来说，Linux 都在 /proc/[pid]/root 处创建了一个链接（详细信息请参考 proc 文件系统手册）。因而，容器内的文件就可以通过 /proc/[pid]/root 在主机中访问。

你可以执行下面的命令，查询容器的 PID，进而再查询 bash 的 uprobe 列表：

复制代码

```
# 查询容器进程在主机命名空间中的PID
PID=$(docker inspect -f '{{.State.Pid}}' bash)

# 查询uprobe
sudo bpftrace -l "uprobe:/proc/$PID/root/usr/bin/bash:*"

# 跟踪bash:readline的结果
sudo bpftrace -e "uretprobe:/proc/$PID/root/usr/bin/bash:readline { printf(\"U
```

如何使用 eBPF 阻止容器的异常行为？

了解了如何分析容器中的安全问题之后，我们再来看看如何阻止容器中的异常行为。

在 eBPF 之前，其实已经有很多的机制来阻止容器的异常行为，比如 AppArmor、Seccomp、Capabilities 等。这些机制虽然好用，但它们的缺点也很明显，那就是它们都需要预先配置好相关的安全策略。在检测到安全隐患之后，更新策略到策略生效中间总有一定的延迟。

而 eBPF 则可以在检测到安全隐患时，实时去触发安全策略的执行。比如，Linux 5.3 版本中新增的辅助函数 bpf_send_signal() 可以直接把有安全隐患的进程杀死，而 Linux 5.7 新增的 LSM 插桩则可以扩展 Linux 安全模块的审计和策略执行（想了解 LSM 钩子的详细信息，你可以参考内核头文件 include/linux/lsm_hooks.h）。在网络安全方面，XDP 程序还可以在网络协议栈之前丢弃数据包，减少非法网络请求对系统资源的消耗。

以上一小节的二进制命令执行为例，你可以在 bpftrace 中调用 signal() 函数，给当前进程发送特定信号：

复制代码

```
tracepoint:syscalls:sys_enter_execve
/comm == "bash"/ {
  $task = (struct task_struct *)curtask;
  $cname = $task->nsproxy->uts_ns->name.nodename;
  printf("Killing shell command in container %s: %s ", $cname, $pidns, comm);
 signal(9); /* SIGKILL */
}
```

这段代码中的具体内容含义如下：

/comm == "bash"/ 表示对进程名称进行过滤，只处理 Bash 进程；

signal(9) 表示向进程发送 SIGKILL 信号，即杀死进程。

加入与上一小节中相同的头文件，然后把它保存到 block-container-shell.bt 文件（你也可以在 GitHub 上找到完整代码）中，然后运行下面的命令来执行：

复制代码

```
sudo bpftrace --unsafe block-container-shell.bt
```

接着，回到容器终端，执行任意命令都会失败，并且失败信息都是 Killed 。

需要注意，这种直接杀死进程的方法实际上比较危险，如果出现误杀，可能会导致大量进程都无法正常启动。所以 bpftrace 要求你加上 --unsafe选项后才可以正常运行。

小结

今天，我带你一起梳理了 eBPF 的安全能力，并以容器应用为例，带你用 eBPF 分析并阻止了容器中的命令执行。

eBPF 所支持的 kprobe、uprobe、tracepoint 等各类探针已经非常全面地涵盖了从应用到内核中的各类安全问题相关的跟踪点，再配合其开销低、实时性好，且不需要修改并重新编译内核和应用等特性，使 eBPF 特别适合用于安全事件的动态监测和实时分析诊断。对于安全策略的执行，eBPF 也已经支持了 LSM 钩子、向进程发送信号、丢弃网络数据包等各类操作。

在这一讲的最后，我想提醒你：既然 eBPF 提供了这么强大的功能，在用好 eBPF 这些丰富特性的同时，你也要特别留意 eBPF 程序自身的安全性。比如，禁止普通用户和普通容器应用运行 eBPF 程序，而只允许管理员和系统服务运行。对容器来说，你可以利用 Capabilities 禁止普通容器的 `CAP_PERFMON`、`CAP_BPF`和 `CAP_SYS_ADMIN`等权限。

思考题

最后，我想邀请你来聊一聊：

1. 在了解 eBPF 之前，你是如何监测、分析和阻止容器的安全问题的？
2. 在阻止容器 Bash 命令执行的案例中，除了杀死进程之外，还有哪些其他的方法？

期待你在留言区和我讨论，也欢迎把这节课分享给你的同事、朋友。让我们一起在实战中演练，在交流中进步。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 | 网络跟踪：如何使用 eBPF 排查网络问题？

下一篇 12 | 高性能网络实战（上）：如何开发一个负载均衡器？

精选留言 (4)

写留言

莫名
2022-02-10

另外分享一个经验，结构体 nsproxy 的 net 字段（代表进程所在的网络命名空间）在追踪网络包在宿主机 -> 容器传递过程时尤其有用，可以比较清楚的看到网络包从宿主机的网络命名空间传递到容器独立的网络命名空间之中，协助更好的理解容器网络模型。（把网络设备名字也打印出来会更好。）

...

展开

作者回复: 非常棒的示例，谢谢分享

2

莫名
2022-02-10

1、曾使用过 sysdig，老版本通过插入内核模块的方式进行安全审计。后来 sysdig 支持了 eBPF driver，主要通过追踪系统调用分析可能的安全隐患。sysdig eBPF driver 实现比较简单，一共十几个 program，统一放在 probe.c 源文件，里面的思路借鉴下还是不错的。
2、觉得需要在 bash 自身上做手脚，比如把 bash 软链接到一个脚本文件，记录下 bash 执行记录，然后 直接 exit。或者修改 bash 配置文件 .bashrc，文件末尾直接 exit。两种...

展开

作者回复: 1. 谢谢分享sysdig的使用经验！
2. 嗯嗯，这是一种不错的方法。或者考虑极端一些，完全禁止掉Bash（比如所有进程都跑在容器中，容器内不允许安装任何SHELL）。

2

写点啥呢
2022-02-09

本节课是关于eBPF在安全方面的使用，微博上看到有人发的这个rookit：https://weibo.com/tv/show/1034:4718242451882158?from=old_pc_videoshow，因此我对eBPF如何保证自身安全性很好奇，想请教下老师对于系统来说eBPF是一个两面利器，它自身设计与实

现，还有在使用中该如何注意避免引入安全问题呢？

...

展开

作者回复: 你说的非常正确，eBPF如果被恶意应用利用，也同样会带来不可预料的损失。所以，在系统维护中，还是需要对进程的权限有所限制。比如，不要用root用户运行进程、不要给容器特权或者ADMIN capability等等。

1

Geek_89541f

2022-02-09

请问我在阿里云的虚拟机（centos系统，内核版本4.18)中使用ip link加载xdp程序，若指定xdpdrv模式会报错，xdpgeneric可以。这是为啥？虚拟网卡只能使用generic模式吗？这样性能不能满足生产需求。

作者回复: xdpdrv模式需要网卡驱动的支持的，具体的驱动支持情况可以参考 https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#xdp

共 2 条评论